FACULTY OF ELECTRONICS AND INFORMATION TECHNOLOGY
TELECOMMUNICATION DEPARTMENT

(Msc- 2ND Semester)

Adaptive Signal Processing

Final Project Report

Submitted By:

Miss. Roshani Thaware (317616)

Mr. Erua Chijioke Nnanna (317608)

Warsaw, Poland

INTRODUCTION:

When the architecture of a convolutional neural network is established, the next step is to train this model. When a model is trained, we are basically trying to solve the problem of algorithm optimization. This so-called optimization is applied to the various arbitrary weights assigned to several neurons or neural nodes in the defined architecture. In training these weights, we are only trying to modify them in such a way that they are able to reach their optimal value. The way these weights are going to be updated generally depends on the type of optimization method or optimizer that will be used. The most widely used optimizer is the stochastic gradient descent (SGD). The primary objective of this optimizer (SGD) is to minimize the loss function (generally referred to as the error function).

The SDG will be assigning the weights to the neural nodes in such a way as to make the loss function as close to zero as possible. This loss function is actually the square of the difference between the actual value and the predicted value that a particular output node generated after a single iteration process. As earlier said, the SDG minimizes this loss function to make our neural network make better and more accurate predictions.

Through this repeated modification of the weight at each epoch, our model experiences what we call machine learning.

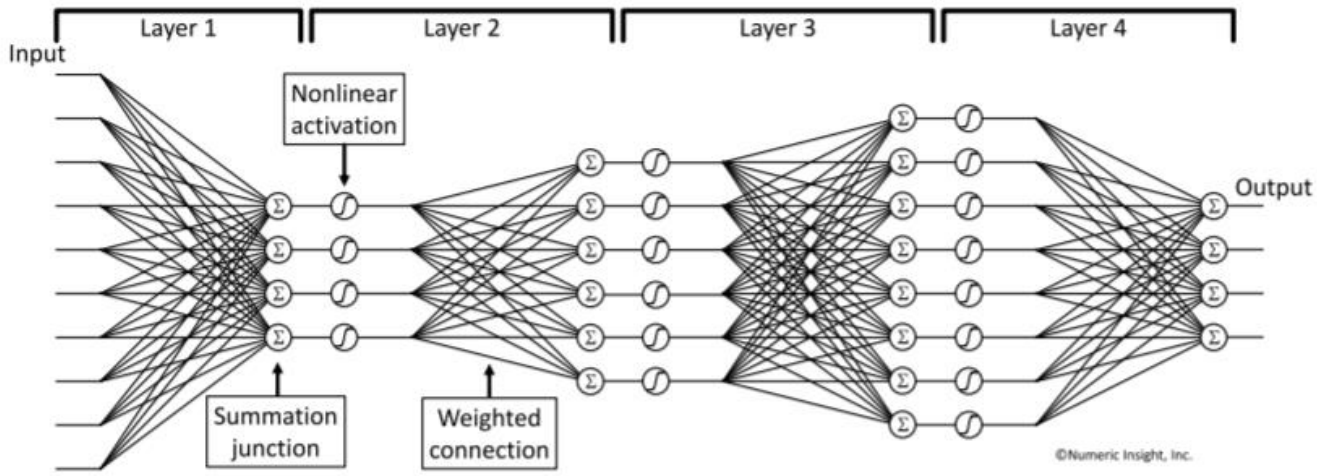


Fig: Image showing an example of a multi-layered neural network that can be used to associate an input consisting of 10 numbers with one of 4 decisions or predictions.

Image source:
https://www.researchgate.net/publication/266396438_A_Gentle_Introduction_to_Backpropagation

Now recall that when a neural model is first initialized, it is set with randomly selected weights and at the end of the output node, there will be a unique output for each input.

- Once this output is generated, a comparison is carried out between the produced output and the desired output. At this point, the model goes further to compute the gradient of the loss function with respect to each of the weights that have been set and uses the result produced to update and replace the prior weight that was used.
- This process is carried out for each weight in the model and since the weights are different, the derivation process will produce a different new weight result. As these

weights are gradually been updated after each epoch, they get closer to their optimized value, and at the same time, the SDG works to minimize the loss function.

- This updating of the weight is essentially what we mean when we say that the model is learning. The model is learning what values to assign to each weight based on how those incremental changes are affecting the loss function. This process of actually calculating the gradient or taking the derivative of the loss function with respect to each weight in the model is referred to as **Backpropagation Algorithm.**

FUNCTION NOTATION USED IN BACKPROPAGATION ALGORITHM:

How does backpropagation fit into the training process?

After we first forward-propagate our initial data through the network, an output is generated and the loss is calculated for that predicted output based on the true value of the original data. SGD minimizes this loss by calculating the derivative of the loss with respect to each of the weights in the network and uses this derivative to update the weight. This process is repeated over and over again until it finds the minimized loss. Essentially SDG is using backpropagation as a tool to calculate the derivative or the gradient of the loss function. The mathematical process analyzed below shows how backpropagation calculates the gradient of the loss function with respect to these weights. We define the notations used as follows:

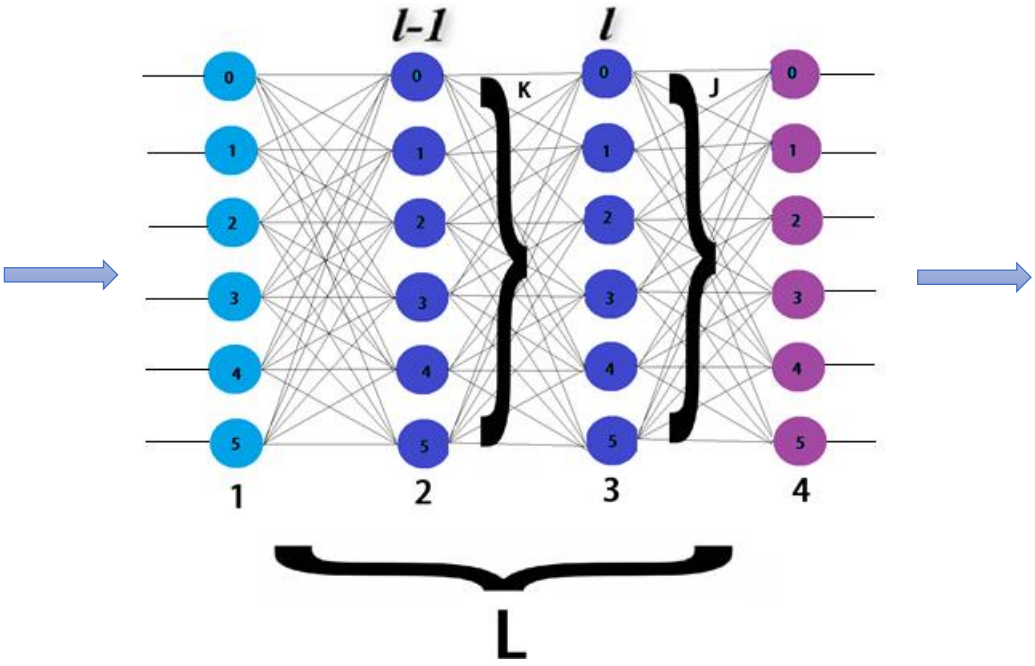


Fig: Neural network with two hidden layers and six nodes.

- ***L*= the number of layers in the architecture of the network.**
- **Layers are indexed as *l = 1, 2, 3, …, L – 1, L***
- **The nodes in a given layer *l* are indexed as *j = 0, 1, 2, …, n - 1***
- **Nodes in the layer *l - 1* are indexed as *k = 0, 1, 2, …, n - 1***
- **y_j = the desired value of node *j* in output layer *L* for a single training sample.**
- **C_0 = Loss function of the network for a single training sample (sum of square errors)**
- **$w_{jk}^{(l)}$ = The weight of the connection that connects node *k* in layer *l - 1* to node *j* in layer *l*.**
- **$w_j^{(l)}$ = The vector that contains all weights connected to node *j* in layer *l* by each node in layer *l - 1*.**

- $z_j^{(l)}$ = **The input for node *j* in layer *l***
- $g^{(l)}$ = **the activation function used for layer *l***
- $a_j^{(l)}$ = **the activation output of node *j* in layer *l***

For the above notations, we focus more on what happens on the network if one training input sample is passed into the network at one time and observing how backpropagation works in that remark. The logic generated for one input sample is then used to generalize what happens when we have more than one input sample passed through the network.

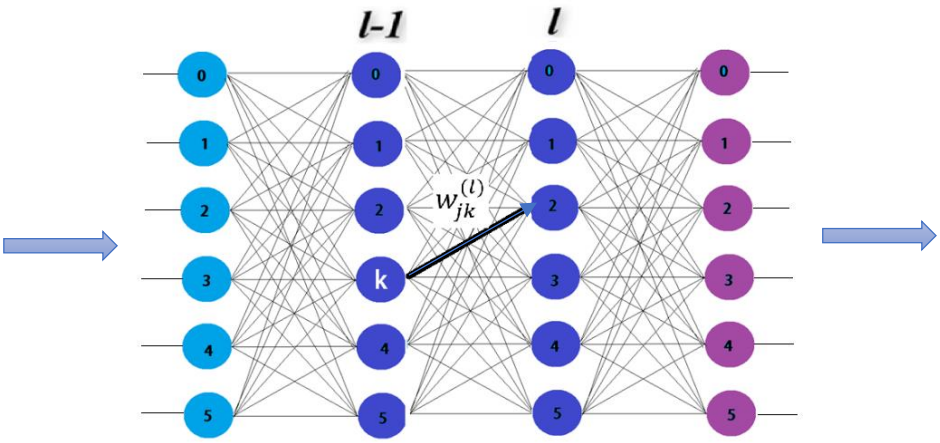


Fig showing $w_{jk}^{(l)}$

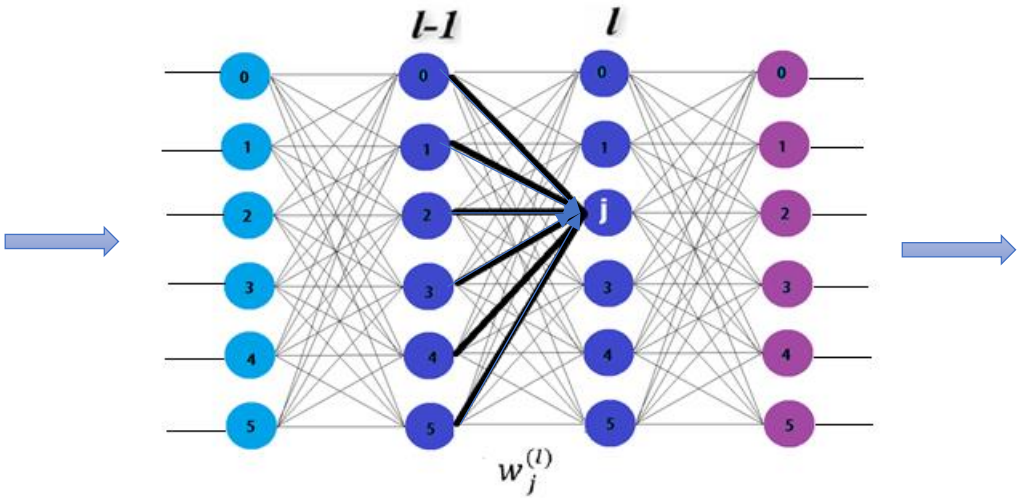


Fig showing $w_j^{(l)}$

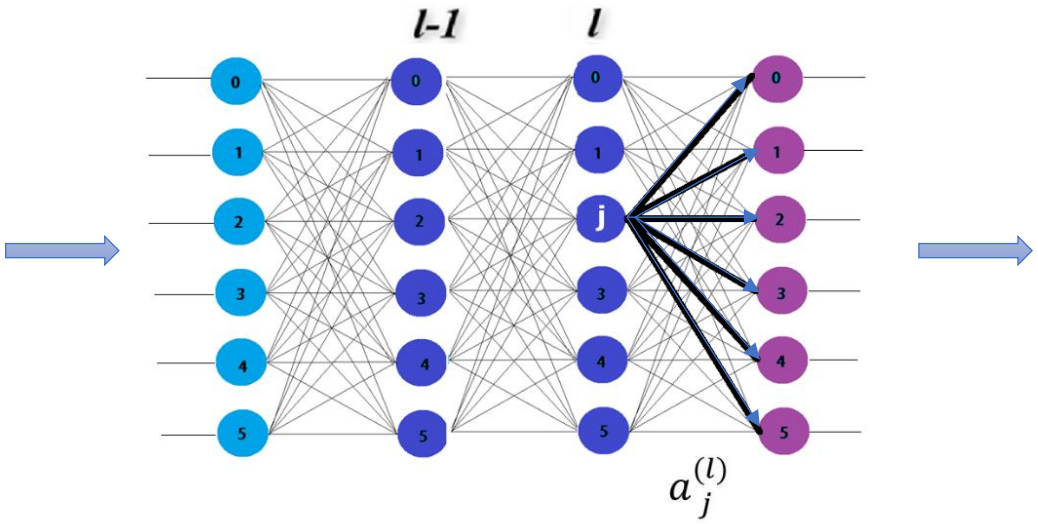


Fig showing $a_j^{(l)}$

Loss C_0 (for a single training sample)

Observe that the expression below

$$(a_j^{(l)} - y_j)^2$$

Shows the square of the difference between the desired output from node j in the output layer l and the activation output in node same node and layer. This can be interpreted as the loss for node *j* in layer *L*. Therefore to calculate the total loss we should sum the squared difference for each node *j* in the output layer *L*. This is express as:

$$C_0 = \sum_{j=0}^{n-1} (a_j^{(l)} - y_j)^2$$

Input $z_j^{(l)}$

Here we look at the input that a particular node *j* within a given layer *l* receives from the previous layer.

We know that the input for the node *j* in layer *l* is the weighted sum of the activation outputs from the previous layer *l - 1*. Considering only a single term gives us the expression below thus:

$$w_{jk}^{(l)} a_k^{l-1}$$

Therefore the input for the given node *j* in layer *l* expressed as:

$$z_j^{(l)} = \sum_{k=0}^{n-1} w_{jk}^{(l)} a_k^{l-1}$$

Activation output $a_j^{(l)}$

We know that the activation output of a given node *j* in layer *l* is the result of passing the input, $z_j^{(l)}$, to whatever activation function we choose to use, say $g^{(l)}$ therefore the activation output of node *j* in layer *l* is expressed as $a_j^{(l)} = g^{(l)}(z_j^{(l)})$

Expressing C_0 as a composition of functions

Recall the definition of C_0 is:

$$C_0 = \sum_{j=0}^{n-l} (a_j^{(l)} - y_j)^2$$

So the loss of a single node *j* in the output layer *L* can be expressed as:

$$C_{0j} = (a_j^{(l)} - y_j)^2$$

We see that the loss from node *j* termed C_{0j} is a function of the activation output of node *j* in layer *L*.

Therefore we can express C_{0j} as a function of $a_j^{(l)}$ as:

$$C_{0j}(a_j^{(l)})$$

Observe from the definition of C_{0j} that C_{0j} also depends on y_j. Since y_j is a constant, we only observe C_{0j} as a function of $a_j^{(l)}$, and y_j as a parameter that helps define this function. The activation output of node *j* in the output layer *L* is a function of the input for node *j*. From an earlier observation we can express this as:

$$a_j^{(l)} = g^{(l)}(z_j^{(l)})$$

The input for node *j* is a function of all the weights connected to node *j* so we can express $z_j^{(l)}$ as a function of $w_j^{(l)}$ as:

$$z_j^{(l)}(w_j^{(l)})$$

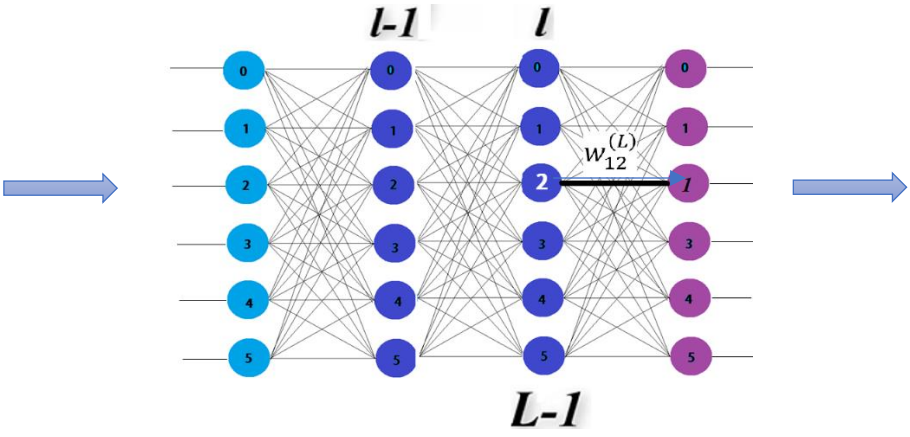
Therefore, $$C_{0j} = C_{0j}(a_j^{(l)}(z_j^{(l)}(w_j^{(l)})))$$

So we can see that C_0 is a composition of functions. We know that $C_0 = \sum_{j=0}^{n-1} C_{0j}$

Therefore applying the same principle, we see that the total loss generated by the neural network for a given single individual input is equally a composition of different functions. This is useful in order to understand how to differentiate C_0

To differentiate a composition of function we make use the chain rule of differentiation.

Derivation of loss with respect to weights



Let's look at the single weight that connect node *2* in layer *L – 1* to node *l* in layer *L*.

This weight is denoted as $w_{12}^{(L)}$. Differentiating the loss with respect to weight $w_{12}^{(L)}$ is given below thus:

$$\frac{\partial C_0}{\partial w_{12}^{(L)}}$$

Since C_0 depends on $a_1^{(L)}$, and $a_1^{(L)}$ depends $z_1^{(L)}$, and $z_1^{(L)}$ depends on $w_{12}^{(L)}$ then the chain rule tells us that to differentiate C_0 with respect to $w_{12}^{(L)}$, we take the product of derivative of the composite function which is expressed as:

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right)$$

Breaking down the mathematical expression on the RHS of the equation given above, we have thus:

The first term: $\frac{\partial C_0}{\partial a_1^{(L)}}$

We know that $C_0 = \sum_{j=0}^{n-l} (a_j^{(l)} - y_j)^2$

Therefore $\frac{\partial C_0}{\partial a_1^{(L)}} = \frac{\partial}{\partial a_1^{(L)}}\left(\sum_{j=0}^{n-l} (a_j^{(l)} - y_j)^2\right)$

Expanding the sum we see

$$\frac{\partial}{\partial a_1^{(L)}}\left(\sum_{j=0}^{n-l} \left(a_j^{(l)} - y_j\right)^2\right)$$

$$= \frac{\partial}{\partial a_1^{(L)}}\left((a_0^{(L)} - y_0)^2 + (a_1^{(L)} - y_1)^2 + (a_2^{(L)} - y_2)^2 + (a_3^{(L)} - y_3)^2\right)$$

$$= \frac{\partial}{\partial a_1^{(L)}}\left(a_0^{(L)} - y_0\right)^2) + \frac{\partial}{\partial a_1^{(L)}}((a_1^{(L)} - y_1)^2) + \frac{\partial}{\partial a_1^{(L)}}((a_2^{(L)} - y_2)^2) + \frac{\partial}{\partial a_1^{(L)}}((a_3^{(L)} - y_3)^2)$$

$$= 2(a_1^{(L)} - y_1).$$

So the loss from the network for a single input sample will respond to a small change in the activation output from node *1* in layer *L* by an amount equal to two times the difference of the activation output a_1 for node *1* and the desired output y_1 for node *1*.

The second term $\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}$

We know that for each node *j* in the output layer *L* we have

$$a_j^{(L)} = g^{(L)}(z_j^{(L)})$$

And since *j = 1* we have

$$a_1^{(L)} = g^{(L)}(z_1^{(L)})$$

$$\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} = \frac{\partial}{\partial z_1^{(L)}} g^{(L)}(z_1^{(L)})$$

$$= g'^{(L)}(z_1^{(L)})$$

Third term $\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}$

We know that, for each node *j* in the output layer *L* we have

$$z_j^{(L)} = \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$$

Since *j = 1*, we have

$$z_1^{(L)} = \sum_{k=0}^{n-1} w_{1k}^{(L)} a_k^{(L-1)}$$

Therefore

$$\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} = \frac{\partial}{\partial w_{12}^{(L)}} \left(\sum_{k=0}^{n-1} w_{1k}^{(L)} a_k^{(L-1)}\right)$$

And expanding the sum we have,

$$\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} = \frac{\partial}{\partial w_{12}^{(L)}} \left(\sum_{k=0}^{n-1} w_{10}^{(L)} a_0^{(L-1)} + w_{11}^{(L)} a_1^{(L-1)} + w_{12}^{(L)} a_2^{(L-1)} + \cdots + w_{15}^{(L)} a_5^{(L-1)}\right)$$

$$= \frac{\partial}{\partial w_{12}^{(L)}} w_{10}^{(L)} a_0^{(L-1)} + \frac{\partial}{\partial w_{12}^{(L)}} w_{11}^{(L)} a_1^{(L-1)} + \frac{\partial}{\partial w_{12}^{(L)}} w_{12}^{(L)} a_2^{(L-1)} + \cdots + \frac{\partial}{\partial w_{12}^{(L)}} w_{15}^{(L)} a_5^{(L-1)}$$

$$= a_2^{(L-1)}$$

So the input node for *1* in layer *L* will respond to change in weight $w_{12}^{(L)}$ by an amount equal to the activation output for node *2* in the previous layer *L - 1*.

Combining terms

Combining all terms we have

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right)$$

$$= 2(a_1^{(L)} - y_1)(g'^{(L)}(z_1^{(L)}))(a_2^{(L-1)})$$

Conclusion

So now we have seen how to calculate the derivative of the loss with respect to one individual training sample. To calculate the derivative of loss with respect to the this same particular weight w_{12}, for all and *n* training samples, we calculate the average derivate of the loss function over all *n* training samples.

This can be expressed as

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = 1/n \sum_{i=0}^{n-1} \frac{\partial C_i}{\partial w_{12}^{(L)}}$$

We would then further apply this mathematical procedure for each and every known weight in the neural network to calculate the derivative of the loss (C) wrt each weight.

Derivation of loss with respect to the activation outputs

Recall that we can calculate the gradient of the loss function with respect to any weight in the network. The weight we choose to work with to explain the idea was $\boldsymbol{w_{12}^{(L)}}$, and we saw that:

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = (\frac{\partial C_0}{\partial a_1^{(L)}})(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}})(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}})$$

Suppose we choose to work with the weight that is not in the output layer like, $\boldsymbol{w_{22}^{(L-1)}}$ then the gradient of the loss with respect to this weight would be

$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L)}}{\partial w_{22}^{(L-1)}}\right)$$

The second and the third term on the right hand side would be calculated in exact same way as we saw for $\boldsymbol{w_{12}^{(L)}}$. The first term on the right hand side $\frac{\partial C_0}{\partial a_2^{(L-1)}}$, will not be calculated in the same way as before. We need to understand how to calculate this term in order to calculate the gradient of the loss function with respect to any weight that is *not* in the output layer.

The calculation of this term will be our focus.

Set up is as follows:

We are going to show how we can calculate the derivative of the loss function with respect to the activation output for any node that is not in the output layer. Let us look at a single activation output for layer *2* in layer *L - 1*. This is denoted as $\boldsymbol{a_2^{(L-1)}}$. The derivative of loss, $\boldsymbol{C_0}$ with respect to this particular activation output $\boldsymbol{a_2^{(L-1)}}$ is denoted as $\frac{\partial C_0}{\partial a_2^{(L-1)}}$. Observe that the node *j* in *L*, the loss $\boldsymbol{C_0}$ depend on $\boldsymbol{a_j^{(L)}}$ and $\boldsymbol{a_j^{(L)}}$ depend on $\boldsymbol{z_j^{(L)}}$. The term $\boldsymbol{z_j^{(L)}}$ depends on all the weight connected to node *j* from the previous layer, *L – 1*, as well as the activation output from *L – 1*. So $\boldsymbol{z_j^{(L)}}$ depends on $\boldsymbol{a_2^{(L-1)}}$.

Recall that applying the chain rule of differentiation, we differentiate the loss ($\boldsymbol{C_0}$) with respect to $\boldsymbol{a_2^{(L-1)}}$, we compute the derivative of each function making up the composite function and then take the product of them. This derivative can be expressed as:

$$\frac{\partial C_0}{\partial a_2^{(L-1)}} = \sum_{j=0}^{n-1} \left(\left(\frac{\partial C_0}{\partial a_j^{(L)}} \right) \left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \right) \left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} \right) \right)$$

This equation look almost identical to the equation with obtained for the derivative of the loss with respect to a given weight. Recall that this previous derivative with respect to a given weight as expressed as:

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}} \right) \left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} \right) \left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}} \right)$$

The two differences between the derivative of loss with respect to an activation output and the derivative of the loss with respect to a weight are:

a) First the summarisation operation

b) The last term on the RHS of the expression

The reason for the summarisation here is due to the fact that a change in one activation output in the previous layer is going to affect each node *j* in the following layer *L*, so we need to some of this effects. We can see that the first and second term on the right hand side of the equation are the same as the first and second term in the last equation with regard to $\boldsymbol{w_{12}^{(L)}}$ when *j =1*. Since we have already gone through the work to find out how to calculate these two derivative, we will only focus on breaking down the third term.

Third term $\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}$

Recall that for each given node *j* in layer *L* we have that

$$z_j^{(L)} = \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$$

Therefore,

$$\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} = \frac{\partial}{\partial a_2^{(L-1)}} \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$$

Expanding the sum, we have

$$\frac{\partial}{\partial a_2^{(L-1)}} \sum_{k=0}^{n-1} w_{jk}^{(L)} a_k^{(L-1)}$$

$$= \frac{\partial}{\partial a_2^{(L-1)}} \left(w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + \cdots + w_{j5}^{(L)} a_5^{(L-1)} \right)$$

$$= \frac{\partial}{\partial a_2^{(L-1)}} w_{j0}^{(L)} a_0^{(L-1)} + \frac{\partial}{\partial a_2^{(L-1)}} w_{j1}^{(L)} a_1^{(L-1)} + \frac{\partial}{\partial a_2^{(L-1)}} w_{j2}^{(L)} a_2^{(L-1)} + \ldots + \frac{\partial}{\partial a_2^{(L-1)}} w_{j5}^{(L)} a_5^{(L-1)}$$

$$= w_{j2}^{(L)}$$

So the input for any node *j* in layer *L* will respond to change in $\frac{\partial}{\partial a_2^{(L-1)}}$ by an amount equal to the weight connecting node *2* and node j in the two different layers.

Combining terms we have thus,

$$\frac{\partial C_0}{\partial a_2^{(L-1)}} = \sum_{j=0}^{n-1} \left(\left(\frac{\partial C_0}{\partial a_j^{(L)}} \right) \left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \right) \left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}} \right) \right)$$

$$= \sum_{j=0}^{n-1} (2(a_j^{(L)} - y_j)(g'^{(L)}(z_j^{(L)}))(w_{j2}^{(L-1)}))$$

Moving further, we can use this obtained result to compute and find the gradient of the loss with respect to any known weight that is linked to node 2 in layer L – 1, as seen, for example with the following equation.

$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}} \right) \left(\frac{\partial a_2^{(L)}}{\partial z_2^{(L-1)}} \right) \left(\frac{\partial z_2^{(L)}}{\partial w_{22}^{(L-1)}} \right)$$

Note that to find the derivative of the loss function with respect to this same particular activation output, $a_2^{(L-1)}$, for all *n* training samples, we calculate the average derivative of the loss function over all *n* training samples. This can be expressed as:

$$\frac{\partial C}{\partial a_2^{(L-1)}} = 1/n \sum_{i=0}^{n-1} \frac{\partial C_i}{\partial a_2^{(L-1)}}$$

RESULT ANALYSIS

In the stage we will analyse the backpropagation algorithm we designed and generated using a python algorithm and then show the different output signals we produced for a randomly selected input signal. The neural network that we designed on the code has four different layers (one input layer, two hidden layers, and one output layer). The development of the back propagation algorithm can be discussed in xx stages as follows:

- Generation of random signals (inputs & outputs) using a non-linear model.
- Scaling of the randomly generated signals
- Designing the neural network
  - Generating random weights in the network
  - Definition of the Activation function
  - Feed-forward propagation
  - Backpropagation
- Training the neural network with the input signals and printing of results

To **generate the random signals (inputs and outputs**), we define a function called function generator as seen in the source code image below. Samples_A represents the randomly selected amplitude of the carrier and message signals (A_c and A_m) respectively. These values of Samples_A serve as input to the non-linear function called "model". Model is a nonlinear function which incorporates sinusoidal functions as its coefficients. Based on these model, a corresponding output s data is produced for each of the randomly selected input signal. This generated signals serve as our training data set.

```
#Defining our training dataset
# Input = A = (Ac, Am),
# Ac = Amplitude of the carrier Signal
# Am = Amplitude of message signal
# y = Output = Amplitude of Modlated Signal
```

```
import numpy as np
```

```
# Define the data generator to generate random input signals which will work with the "model" defined here to produce outputs

def dataGenerator(r, c):
    samples_A = np.random.randint(1,10, (r, c))
    samples_y = np.array([], dtype = float)
    wT = 6
    for rows in samples_A:
        #t = rows[0]*10 + 5*rows[1]**2 + 4  # model y = 10A1 + 5A2^2 + 4
        model = rows[0]* np.sin(wT) + rows[1]**2*np.cos(wT) + 4
        samples_y = np.append(samples_y, model)

    samples_A = samples_A.astype(np.float64)
    samples_y = samples_y.astype(np.float64).reshape(r, 1)
    return (samples_A, samples_y)
```

```
solution = dataGenerator(10,2)
A = solution[0]
y = solution[1]
print('Value for A:\n', A)
print('Value for y:\n', y)
```

Fig: Image showing the generation of random input and output signals

After the signals are generated, we scale them. This **scaling of generated signals** is done by dividing all the entries in the first column of the matrix A with the max value contained in that column. The scaling of the output signal is done by dividing the entries of the output matrix by 100.

```
# scale x and y by dividing x with the max of the input array and dividing y by 100
A = A/np.amax(A, axis=0)
y = y/100

print(A)
print(y)
```

Fig: Image showing the scaling of the random input and output signals

To **design the neural network**, we first establish the number of layers and initialize the entries each array should contain. We made use of object oriented programming to design this neural network. As seen from the source code below, the max number of entries in an array of the input node is 2 ([(A_c, A_m)]), the number of entries in an array of the output node is 1 and that of the hidden node is 3. Since in the first stage of training a neural network, we need to assign random values to the weights that make-up the network as seen on the last section of the image below-

```
#providing initial values for the layers of the neural network we use OOP
class NeuralNetwork(object):
    def __init__(self):
#parameters for the Four layers of our neural network
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize1 = 3
        self.hiddenSize2 = 3


#weights: Assigning random weights to the first input
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize1) # (2xy3) weight matrix moving from input to hidden layer1
        self.W2 = np.random.randn(self.hiddenSize1, self.hiddenSize2) # (3x3) weight matrix moving from hiddenlayer1 to hidden la
        self.W3 = np.random.randn(self.hiddenSize2, self.outputSize) # (3x1) weight matrix from hidden2 to output layer
```

Fig: code section showing randomly selected initial weights.

We then **define an activation function** to be used in our neural network. The activation function manages the incoming weighted sum and performs a check on it in such a way that if the value is too big, it reduces it and vice versa. We made use of the sigmoid function as the activation function. The sigmoid acts by placing any number passed through it in between the two extremes of 0 and 1. From the image we can see return values if the derivative is true and false respectively.

```
#The Activation function which converts a number to a probability i.e between 0 and 1
    def sigmoid(self, s, deriv=False): #in backpropagation we use the derivative of the sigmoid to find the tangent slope
        if (deriv == True):
            return s * (1 - s) #derivative of the sigmoid
        return 1/(1 + np.exp(-s))
```

Fig: showing the activation function (sigmoid function)

The next step is to carry out **forward propagation** of the data through the network. This is done by multiplying the input signal (A) with the weight (self.W1) connecting the input node and the first hidden layer. Since the activation function is located at the node of the first hidden

layer, it acts upon the incoming weighted sum and the output produced serves as an input to the next layer. This multiplication of weights and inputs is done in the forward direction until we reach the output node. Note how the activation function is placed at each node so that it can act on each input coming into the node and produce an activation output.

```
#forward propogation through the Neural Network
    def feedForward(self, A):
        self.z = np.dot(A, self.W1) #The dot product of input (A) and first set of weights (2x3) matrix above
        self.z2 = self.sigmoid(self.z) #output produced after activation function is applied to the incoming input (self.z)

        self.z12 = np.dot(self.z2, self.W2) #The dot product of input to hidden layer 1 and second set of weights (3x3) matrix ab
        self.z22 = self.sigmoid(self.z12) #output produced after activation function is applied to the incoming input (self.z12)

        self.z3 = np.dot(self.z22, self.W3) #dot product of hidden layer 2 (output of the sigmoid fxn above) and third set of we
        output = self.sigmoid(self.z3) #Applying the sigmoid fxn on the fourth layer we get the output
        return output
```

Fig: Feedforward propagation in the neural network

After the feed forward propagation is done, the next thing is to carry out **backpropagation** and for us to achieve this, we need to first determine how far away the output we produced is from the actual output that was randomly selected initially at the beginning of this procedure. The difference between the actual output and the predicted output gives us what is termed output error. We then defined a function which is called output_delta which is the product of the output error and the derivative of the sigmoid function. After each iteration, the weights W1, W2, and W3 are updated and this updating continues until a weight combination that gives the least possible error is attained. The updating of the weight is seen in the last three lines of the image below.

```
#backward propogation through the neural network
    def backward(self, A, y, output):
        self.output_error = y - output # error in output
        self.output_delta = self.output_error * self.sigmoid(output, deriv=True)

        self.z22_error = self.output_delta.dot(self.W3.T)#z22 error: how much our hidden layer 1 weights contribute to output er
        self.z22_delta = self.z22_error * self.sigmoid(self.z22, deriv=True)


        self.z2_error = self.z22_delta.dot(self.W2.T) #z2 error: how much our hidden layer 2 weights contribute to output error
        self.z2_delta = self.z2_error * self.sigmoid(self.z2, deriv=True) #applying derivative of sigmoid to z2 error

        self.W1 += A.T.dot(self.z2_delta) # adjusting first set (input -> hidden) weights
        self.W2 += self.z2.T.dot(self.z22_delta) # adjusting second set (hidden -> hidden) weights
        self.W3 += self.z2.T.dot(self.output_delta) # adjusting third set (hidden -> output) weight
```

Fig: Function definition of the back propagation stage of the algorithm

Next, the neural network is trained. To carry out the training process, we defined a function called “train” which takes two arguments, the input and output training data signal sets. The training process comprises the feed-forwarding stage and the backpropagation stage of the whole algorithm. It is to be noted that the network is fully trained only when it minimizes the error so well that our predicted output is close enough to the expected output.

```
#Training the neural network
    def train(self, A, y):
        output = self.feedForward(A)
        self.backward(A, y, output)
```

Fig: Defining the function for training of the neural network

The image below shows the range or number of iterations the network should be trained with. As seen from the image the network is going to be trained for 1000 times but as a result of the conditional statement we attached to this section of the code, only 100 output loss function will be generated. Therefore for every 1000 times the network is trained, 100 loss functions will be trained. We can increase the number of times the network is to be trained by changing the range as seen in the code. If for example we train the neural network 2000 times, we are expected to get 20 loss functions printed accordingly. So as the training range increases, so is the loss function printed.

```
NN = NeuralNetwork()
losses_list = []
iter_list = []
for i in range(1000): #trains the neural network (NN) 1000 times
    if (i % 100 == 0):
        losses_list.append(np.mean(np.square(y - NN.feedForward(A))))
        iter_list.append(i)
        print("Loss: " + str(np.mean(np.square(y - NN.feedForward(A)))))
    NN.train(A, y)

print("Input: " + str(A))
print("Actual Output: " + str(y))
print("Loss: " + str(np.mean(np.square(y - NN.feedForward(A)))))
print("\n")
print("Predicted Output: " + str(NN.feedForward(A)))
```

Fig: Training and result extraction from the neural network

We also added a graphical section in the last section of the source code which will graphically illustrate how the neural network learns as the loss after each itteration reduces

```
plt.figure(figsize=(8,6))
plt.plot(iter_list,losses_list)
plt.xlabel('Number of iterations')
plt.ylabel('Loss')
plt.title('Loss against number of iterations')
plt.grid()
plt.savefig('loss_graph.png')
```

Fig: plotting of the loss function values against the number of iterations

OUTPUT ANALYSIS:

Since examining the convergence or minimization of the loss function for large iterations will be tedious, we inserted a conditional statement in such a way that for every “n” number of iterations, the neural network will print “n/100” loss function value and then equally print the mean of the “n” number of iterations.

We first consider a scenario where 20 randomly selected input signals (amplitude of the message and carrier signal) are used by our model (which is a non-linear function) to generate 20 outputs (Amplitudes of the modulated signal) which is used to train the network for 1000 times.

In this first case, **the non-linear model** we used is:

model = rows[0]* np.sin(wT) + rows[1]2*np.cos(wT) + 4**

When this line of code (samples_A = np.random.randint(1,10, (r, c))) from the definition of the function dataGenerator is executed, we were able to generate 20 random input signals which when used by our model defined above, produces 20 output signals equally. These set of produced outputs with the randomly selected inputs become our training signal datasets. Below is the results obtained after training our model through backpropagation.

Output (Actual and Predicted) result for 20 randomly selected input for our defined Non_linear model.

```
Loss: 0.32837788716115723
Loss: 0.005344219853478856
Loss: 0.0012178822762012019
Loss: 0.0008333779543571402
Loss: 0.0006561530172170761
Loss: 0.0005559322962073134
Loss: 0.0004944157855039982
Loss: 0.000452845610438231
Loss: 0.00042240099313377516
Loss: 0.0023430231231533834
Input: [[0.44444444 0.88888889]
 [0.33333333 0.66666667]
 [0.66666667 0.11111111]
 [0.33333333 0.55555556]
 [0.33333333 0.33333333]
 [0.22222222 0.33333333]
 [0.22222222 0.22222222]
 [0.88888889 0.66666667]
 [0.33333333 0.88888889]
 [0.44444444 0.77777778]
 [0.66666667 0.22222222]
 [1.         0.33333333]
 [0.22222222 0.33333333]
 [0.33333333 0.33333333]
 [0.44444444 1.        ]
 [1.         0.11111111]
 [0.33333333 0.22222222]
 [1.         0.44444444]
 [0.33333333 0.44444444]
 [1.         0.11111111]]
Actual Output: [[0.64333236]
 [0.37727884]
 [0.03283677]
 [0.27166011]
 [0.11803286]
 [0.12082702]
 [0.0728185 ]
 [0.36330806]
 [0.64612652]
 [0.49930682]
 [0.06164188]
 [0.10126793]
 [0.12082702]
 [0.11803286]
 [0.80656131]
 [0.02445431]
 [0.07002435]
 [0.16847985]
 [0.18524478]
 [0.02445431]]
Loss: 0.0007140091278343056
Predicted Output: [[0.67700319]
 [0.43298281]
 [0.04240946]
 [0.30200937]
 [0.11611733]
 [0.11450066]
 [0.06745057]
 [0.39241778]
 [0.67491237]
 [0.56523269]
 [0.06919297]
 [0.1072826 ]
 [0.11450066]
 [0.11611733]
 [0.76155834]
 [0.04331992]
 [0.06845143]
 [0.16956481]
 [0.19289038]
 [0.04331992]]
```

Looking at the results above, we can see how the predicted values of the output signals are in close range with the actual output signals. This is evidently shown in the loss values printed on the figure above. We can see that there is a reduction in the loss as it reduces after every iteration step(see graph of loss against number of iterations below) showing that as the backpropagation algorithm is applied, our neural networks learns after each iteration and gives a good enough prediction.

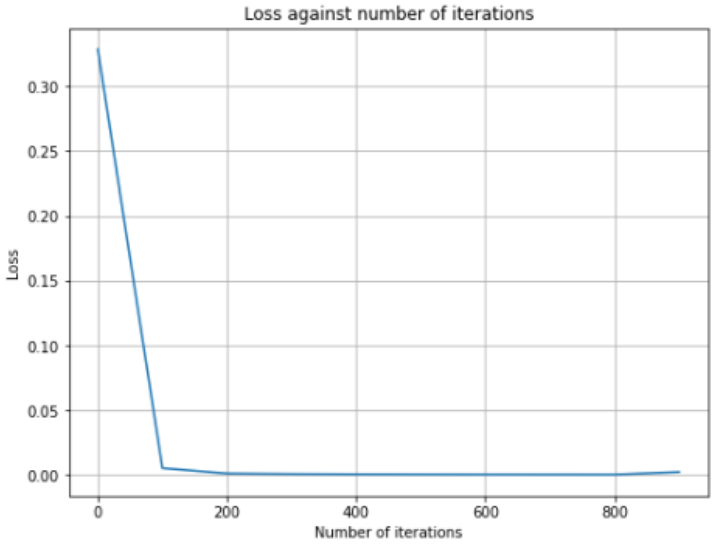


Fig: Graph of loss agains iteration for 20 randomly selected input signals

We now consider a scenario where 100 randomly selected input signals (amplitude of the message and carrier signal) are used by our model (which is a non-linear function) to generate 100 outputs (Amplitudes of the modulated signal) which is used to train the network for 1000 times.

Output (Actual and Predicted) result for 100 randomly selected input for our defined Non_linear model.

Loss: 0.0867534310853802
Loss: 0.04854478990959276
Loss: 0.012235999463299395
Loss: 0.0067178720909981645
Loss: 0.017319443187944563
Loss: 0.006951231729625357
Loss: 0.020722046659893904
Loss: 0.010987943371928377
Loss: 0.010154606911686553
Loss: 0.009417415723172265

Input: [[1. 0.66666667]
[0.44444444 0.55555556]
[0.66666667 0.55555556]
[0.22222222 0.11111111]
[0.11111111 0.77777778]
[0.77777778 0.33333333]
[0.77777778 0.44444444]
[0.33333333 0.11111111]
[0.22222222 0.88888889]
[0.11111111 0.11111111]
[0.66666667 0.55555556]
[0.88888889 0.66666667]
[0.88888889 0.77777778]
[0.11111111 0.33333333]
[0.77777778 0.77777778]
[0.55555556 0.66666667]
[0.44444444 0.44444444]
[0.22222222 0.55555556]
[0.55555556 0.77777778]
[0.22222222 0.44444444]
[0.33333333 0.77777778]
[0.66666667 0.44444444]
[0.77777778 0.44444444]
[0.55555556 0.11111111]
[0.22222222 1.]
[0.55555556 0.11111111]
[0.88888889 0.33333333]
[0.77777778 0.88888889]
[0.55555556 0.66666667]
[0.33333333 0.77777778]
[0.77777778 0.88888889]
[0.77777778 0.33333333]
[0.22222222 0.11111111]
[0.33333333 0.11111111]
[0.55555556 0.11111111]
[0.33333333 0.66666667]
[0.55555556 0.33333333]
[0.11111111 1.]
[0.33333333 0.33333333]
[0.55555556 0.22222222]
[0.44444444 0.55555556]
[1. 0.55555556]
[0.66666667 0.66666667]
[0.11111111 0.22222222]
[0.44444444 0.66666667]
[0.66666667 0.22222222]
[0.55555556 0.44444444]
[0.77777778 0.33333333]
[0.11111111 0.22222222]
[0.55555556 0.66666667]
[0.33333333 0.33333333]
[0.33333333 0.22222222]
[0.22222222 0.11111111]
[0.66666667 0.88888889]
[1. 0.11111111]
[1. 0.22222222]
[0.11111111 0.44444444]
[0.77777778 0.88888889]
[0.22222222 0.44444444]
[0.44444444 0.55555556]
[0.22222222 0.11111111]
[1. 0.22222222]
[0.33333333 1.]
[0.11111111 0.55555556]
[0.66666667 1.]
[0.11111111 0.77777778]
[0.77777778 0.66666667]
[0.44444444 0.11111111]
[1. 0.66666667]
[0.77777778 1.]
[0.55555556 0.33333333]
[0.77777778 0.33333333]
[0.11111111 0.88888889]
[1. 0.33333333]
[0.66666667 0.88888889]
[0.11111111 0.55555556]
[0.44444444 0.11111111]
[0.77777778 0.44444444]
[0.44444444 0.44444444]
[0.55555556 1.]
[1. 0.22222222]
[0.55555556 0.66666667]
[0.77777778 0.88888889]
[0.44444444 0.22222222]
[1. 0.66666667]
[1. 0.22222222]
[1. 0.44444444]
[0.66666667 0.55555556]
[0.55555556 0.55555556]
[0.11111111 0.77777778]
[0.22222222 1.]
[0.88888889 0.88888889]
[0.55555556 0.88888889]
[0.44444444 0.44444444]
[0.88888889 1.]
[0.66666667 0.11111111]
[0.77777778 0.88888889]
[0.55555556 0.55555556]
[0.55555556 0.66666667]
[0.33333333 0.88888889]]

Actual Output:

```
[[0.36051391]
 [0.26886595]
 [0.26327764]
 [0.04401339]
 [0.50768929]
 [0.10685624]
 [0.17406816]
 [0.04121924]
 [0.64892067]
 [0.04680755]
 [0.26327764]
 [0.36330806]
 [0.4881302 ]
 [0.12362117]
 [0.49092436]
 [0.37169053]
 [0.18245063]
 [0.27445426]
 [0.49651267]
 [0.18803894]
 [0.50210098]
 [0.17686232]
 [0.17406816]
 [0.03563093]
 [0.81214962]
 [0.03563093]
 [0.10406209]
 [0.6349499 ]
 [0.37169053]
 [0.50210098]
 [0.6349499 ]
 [0.10685624]
 [0.04401339]
 [0.04121924]
 [0.03563093]
 [0.37727884]
 [0.11244455]
 [0.81494378]
 [0.11803286]
 [0.06443604]
 [0.26886595]
 [0.25489518]
 [0.36889637]
 [0.07561266]
 [0.37448468]
 [0.06164188]
 [0.17965647]
 [0.10685624]
 [0.07561266]
 [0.37169053]
 [0.11803286]
 [0.07002435]
 [0.04401339]
 [0.63774405]
 [0.02445431]
 [0.05325942]
 [0.19083309]
 [0.6349499 ]
 [0.18803894]
 [0.26886595]
 [0.04401339]
 [0.05325942]
 [0.80935547]
 [0.27724842]
 [0.800973  ]
 [0.50768929]
 [0.36610222]
 [0.03842508]
 [0.36051391]
 [0.79817885]
 [0.11244455]
 [0.10685624]
 [0.65171483]
 [0.10126793]
 [0.63774405]
 [0.27724842]
 [0.03842508]
 [0.17406816]
 [0.18245063]
 [0.80376716]
 [0.05325942]
 [0.37169053]
 [0.6349499 ]
 [0.06723019]
 [0.36051391]
 [0.05325942]
 [0.16847985]
 [0.26327764]
 [0.2660718 ]
 [0.50768929]
 [0.81214962]
 [0.63215574]
 [0.64053821]
 [0.18245063]
 [0.79538469]
 [0.03283677]
 [0.6349499 ]
 [0.2660718 ]
 [0.37169053]
 [0.64612652]]
```

Loss:

0.008791392824248

Predicted Output:

```
[[4.27773879e-01]
 [4.17047363e-01]
 [3.95422159e-01]
 [1.41345545e-04]
 [6.68989850e-01]
 [1.60580198e-01]
 [2.99660098e-01]
 [1.87599551e-04]
 [7.44344468e-01]
 [1.00381277e-04]
 [3.95422159e-01]
 [4.43408143e-01]
 [5.29340123e-01]
 [1.34184590e-01]
 [5.63722459e-01]
 [5.11958234e-01]
 [3.06598347e-01]
 [4.29917735e-01]
 [6.25362532e-01]
 [2.96832487e-01]
 [6.61774303e-01]
 [3.04437694e-01]
 [2.99660098e-01]
 [3.38075809e-04]
 [7.90676137e-01]
 [3.38075809e-04]
 [1.53952145e-01]
 [6.67696905e-01]
 [5.11958234e-01]
 [6.61774303e-01]
 [6.67696905e-01]
 [1.60580198e-01]
 [1.41345545e-04]
 [1.87599551e-04]
 [3.38075809e-04]
 [5.50666497e-01]
 [1.68409545e-01]
 [7.89671535e-01]
 [1.61019145e-01]
 [2.89817688e-02]
 [4.17047363e-01]
 [3.70079227e-01]
 [4.87355429e-01]
 [1.91071666e-02]
 [5.34060793e-01]
 [2.94142245e-02]
 [3.06712844e-01]
 [1.60580198e-01]
 [1.91071666e-02]
 [5.11958234e-01]
 [1.61019145e-01]
 [2.64604199e-02]
 [1.41345545e-04]
 [6.95807030e-01]
 [2.09508681e-03]
 [3.41608499e-02]
 [2.84432304e-01]
 [6.67696905e-01]
 [2.96832487e-01]
 [4.17047363e-01]
 [1.41345545e-04]
 [3.41608499e-02]
 [7.89034232e-01]
 [4.26741563e-01]
 [7.63380279e-01]
 [6.68989850e-01]
 [4.63619342e-01]
 [2.47423709e-04]
 [4.27773879e-01]
 [7.45500908e-01]
 [1.68409545e-01]
 [1.60580198e-01]
 [7.43511146e-01]
 [1.47484148e-01]
 [6.95807030e-01]
 [4.26741563e-01]
 [2.47423709e-04]
 [2.99660098e-01]
 [3.06598347e-01]
 [7.76040843e-01]
 [3.41608499e-02]
 [5.11958234e-01]
 [6.67696905e-01]
 [2.81943517e-02]
 [4.27773879e-01]
 [3.41608499e-02]
 [2.82392978e-01]
 [3.95422159e-01]
 [4.06119404e-01]
 [6.68989850e-01]
 [7.90676137e-01]
 [6.33312923e-01]
 [7.17069587e-01]
 [3.06598347e-01]
 [7.21566613e-01]
 [4.90401897e-04]
 [6.67696905e-01]
 [4.06119404e-01]
 [5.11958234e-01]
 [7.40638212e-01]
```

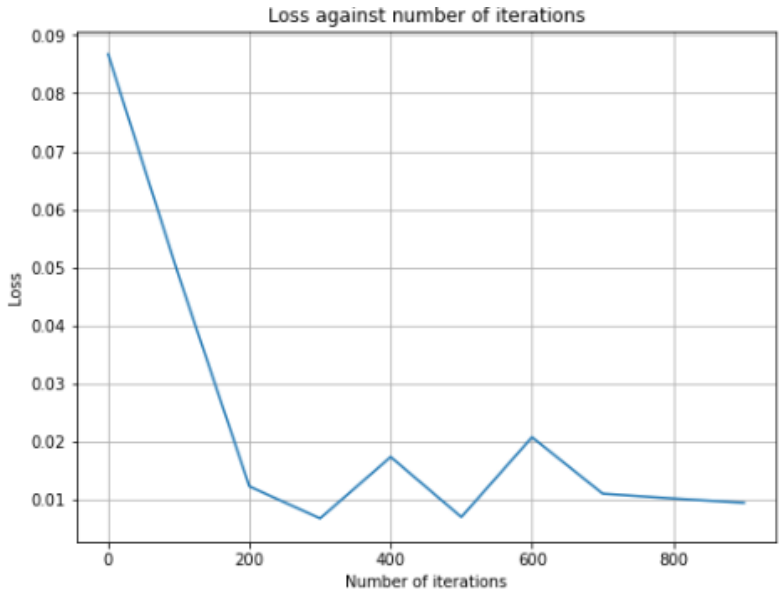


Fig: Plot of the loss function against iterraation for 100 randomly selected signals

Looking at the results above, we can see how the predicted values of the output signals are in close range with the actual output signals. This is evidently shown in the loss values printed above. We can see that there is a reduction in the loss function as it reduces after every iteration step (as seen in the plot of loss against iterration above), showing that as the backpropagation algorithm is applied, our neural networks learns after each iteration and gives a good enough prediction of the actual output. With the above observations we can firmly say that our back propagation Algorithm has been implemented correctly.

References

1 Principles of training multi-layer neural network using backpropagation https://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

2 Durbin, R., & Rumelhart, D. E. (1989). Product units: A computationally powerful and biologically lausible extension to backpropagation networks. Neural Computation, 1, 133-142.

3 Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer Feed-forward Networks are Universal Approximators, Neural Networks, 2, pp. 359-366.

4 How the backpropagation algorithm works http://neuralnetworksanddeeplearning.com/chap2.html

5 BRIDLE, 1989: J. S. Bridle, Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition, in F. Fogelman-Soulié and J. Hérault (eds.), Neuro-computing: Algorithms, Architectures, Springer-Verlag, New York.

6 MINAI, 1990: A. A. Minai and R. D. Williams, Acceleration of BackPropagation through Learning Rate and Momentum Adaptation, International Joint Conference on Neural Networks, vol. 1, January, pp. 676–679, 1990.

7 SAMAD, 1988: T. Samad, "Back-Propagation is significantly", International Neural Network Society Conference Abstracts, 1988.

8 WEIGEND, 1991: A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, BackPropagation, Weight-elimination and Time Series Prediction, in AA.VV. 1991: pp. 857–882.
9 WIDROW, 1985: B. Widrow and S. D. Steams, Adaptive Signal Processing, Signal Processing Series, Prentice-Hall, Englewood Cliffs, NJ, 1985.
10 RUMELHART, 1986c: D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning Internal Representations by Back Propagating Errors, Nature 323: 533–536, in ANDERSON 1988.
11 LAPEDES, 1987: A. Lapedes and R. Farber, Nonlinear Signal Processing Using Neural Networks: Prediction and System Modeling, Los Alamos National Laboratory Report LA-UR-87-2662, 1987.